

An Extended Data-Flow Architecture for Data Analysis and Visualization

Greg Abram and Lloyd Treinish
IBM Thomas J. Watson Research Center
Post Office Box 704
Yorktown Heights, NY 10598

gda@watson.ibm.com and lloyd@watson.ibm.com

Abstract

Modular visualization environments utilizing a data-flow execution model have become quite popular in recent years, especially those that incorporate visual programming tools. However, simplistic implementations of such an execution model are quite limited when applied to problems of realistic complexity, which negate the intuitive advantage of data-flow systems. This situation can be resolved by extending the execution model to incorporate a more complete and efficient programming infrastructure while still preserving the virtues of "pure data-flow". This approach has been used for the implementation of a general-purpose software package, IBM Visualization Data Explorer.

Introduction

Over the last several years a number of software systems that provide "visual programming", which embodied a notion of data-flow, have been implemented [6][15][1][4][11][13][8][9]. They were created under the premise that this paradigm was simple enough for users that are not experienced programmers to build applications. It was further assumed that this approach would greatly simplify the implementation and prototyping of computer graphics, data analysis and visualization systems that are composed of varied and often complex tasks. However, as the visualization community matured and the users of these tools grew in their sophistication, efforts to apply these systems to problems of realistic size and complexity illustrated a number of deficiencies within the typical implementations [12][3]. The challenge from the perspective of developing tools for data analysis and visualization based upon the data-flow paradigm is to preserve the virtues of such an approach while trying to minimize the inherent limitations embodied by the use of a naive data-flow execution model for the visual programs. An outline of the key advantages and disadvantages of the data-flow architecture will establish these points. This will serve as background for a discussion of an implementation that extends the idea of data-flow to include capabilities necessary to support realistic problems, while continuing to supporting its traditional advantages.

Visual Programming and Visualization

The previously mentioned visualization environments incorporate visual programming tools that allow complex systems to be constructed as networks of atomic tasks. For users with an idea of their goals and a basic understanding of the set of provided functions, the construction of sophisticated applications is made simple and intuitive -- programming by plumbing. The process of using these tools closely matches the user's mental model of the computation. In effect, the visual program is simply a graphical representation of the process to be executed.

Visualization applications seem particularly well tailored to the use of a visual programming paradigm. Generally speaking, the atomic operations of the visualization are well-defined and high-level so that sophisticated visualizations can be created by relatively simple networks of tools chosen from a predefined set. Figure 1 illustrates this idea for a program that imports data, computes both an isosurface and a planar mapping, and renders the results in a single image. Visualizations may be parameterized by the incorporation of inputs that relate interactor widgets to network inputs. These inputs may be represented with the visual program simply as modules with no inputs. Instead, their inputs come asynchronously from an associated "input device" or interactor (e.g., a graphics user interface widget such as a slider).

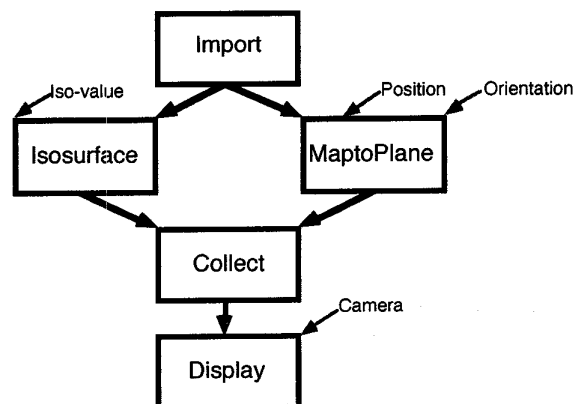


Figure 1. A visual program incorporating two visualization techniques.

(See color plates, page CP-31)

Data-flow Execution

In a true data-flow implementation, all modules are pure functions (i.e., their outputs are fully defined by their inputs). Hence, processes are stateless with no side-effects [2]. An examination of figure 1 leads to such an execution model. Imagine a set of available processes waiting for their inputs from the processes upstream in the network. In figure 1, the *Collect* module waits for inputs from the *Isosurface* and *MapToPlane* modules. When their inputs are received, they run, and when finished they distribute their results to the modules waiting downstream. In figure 1, *Import* would send its results to the waiting *Isosurface* and *MapToPlane* modules. In effect, this execution model is entirely data-driven and top-down: the execution of modules is dependent solely on the passage of data through the system.

Problems with Data-Flow Execution

While this simple data-flow execution model seems a natural mechanism for the execution of visual programs, a closer examination reveals that real-world problems are more complex. In order to function efficiently, it is vital that the system avoid unnecessary work. In general, there are two reasons why modules present in a visual program or (directed acyclic) graph may not need to be executed when their turn comes: 1) their results are not actually required by a result of the network and 2) their inputs are unchanged from the last time the module was executed (i.e., the result will be the same).

Identifying Required Results

In reality, the outputs of a visualization network occur in modules that have side-effects. They produce results outside of the network itself such as the display of images on a workstation or the creation of output files. Unless the result of a module ultimately affects an input to a module that produces a side-effect, that module does not have to be executed (e.g., conditional execution -- see below).

Eliminating modules that are not ancestors of modules with side-effects can be done by pre-processing the network before the actual data-flow network evaluation commences. This is done by traversing the graph bottom-up, beginning at each module known to have side-effects and flagging each module as it is encountered. Once this is complete, modules that have not been flagged do not have to be executed.

Conditional Execution

A much more difficult problem arises when conditionals are incorporated in networks. Conditionals may be used to offer the user a selection among several methods of visualizing a data set. In figure 2, a *Switch* module allows either an isosurface or a mapping plane to be displayed, based on user preference.

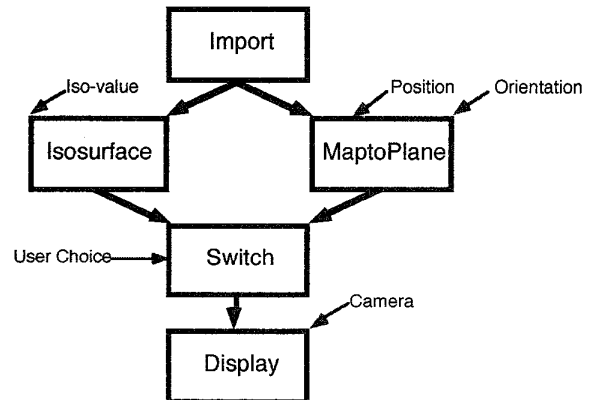


Figure 2. A visual program with a user choosing between two visualization techniques.

In a simple data-flow execution model, this *Switch* module will be executed when its inputs are available (i.e., the results of the *Isosurface* and *MapToPlane* modules, and the value of the selection input). On execution, the *Switch* module chooses whether to pass the *Isosurface* or *MapToPlane* result to the output based on the selection input. In the case of a pure data-flow model both the *Isosurface* and *MapToPlane* modules execute before the decision as to which will actually be used is known. Since both operations can be computationally expensive, the superfluous execution of both of them is very inefficient.

Again, this problem can be handled within the simple data-flow execution module by an examination of the graph prior to execution. In this case, the selection value, which comes from an external source (e.g., an interactor presented to the user) is essentially static, and known a priori. Hence, the selection may be performed by a simple transformation of the graph: excising the *Switch* module altogether, and substituting arcs from the selected source (either *Isosurface* or *MapToPlane*) to each of the modules that, in the original network, received the result of the *Switch* module. This leaves the un-selected module dangling. It and any of its ancestors that are therefore made unnecessary will not be executed.

It should be noted that this approach fails when the selection value is not static (e.g., it is determined elsewhere in the network). Figure 3 illustrates this problem, when the network selects either an isosurface or a set of vector glyphs depending on whether the data are scalar or vector. In this case, the selection value for the *Switch* module cannot be determined before the execution of the graph. Instead, the graph must be evaluated in stages: 1) determine the selection value, 2) determine the necessary input to the *Switch* module and 3) evaluate the remainder of the graph. Since dynamic inputs may themselves be descended from other non-static inputs (e.g., computed in the network), this process may have to be performed repeatedly.

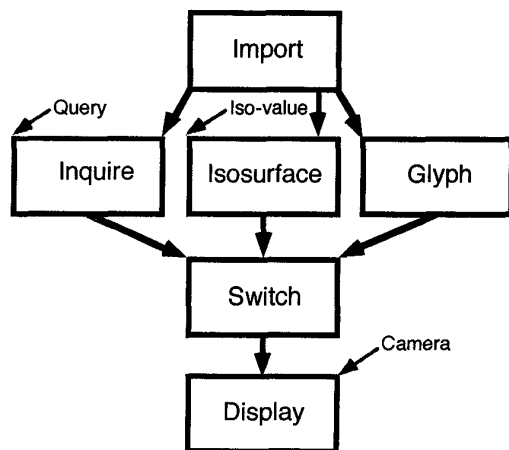


Figure 3. A visual program with computed input to a conditional.

Iterative Re-Execution

Unlike the simple example in figure 1, most real visualizations involve some form of iteration. This may either be direct interaction, where the user is adjusting parameters of the visualization and observing their effect on the resulting images, or animation, in which one or more inputs to the network may vary from frame to frame.

In iterative applications, there are often major parts of the network that are unaffected when input parameters are modified. In figure 2, if the iso-value input to the *Isosurface* module is changed, only the affected modules and their descendents need to be executed. The output of *Import* is not affected by the change. Hence, it can be re-used, which avoids a superfluous access of data on disk. The *MapToPlane* module also does not need to be executed, since its inputs did not change either.

One way to implement this capability is via a caching mechanism for partial results. Instead of immediately re-executing when its inputs arrive, a module may first determine whether its inputs have changed. If they have not changed, it can simply retrieve its results from the cache. Otherwise, the module re-executes, placing its new result into the cache.

Data Explorer Approach

The IBM Visualization Data Explorer (or simply Data Explorer) is a general-purpose software package for data analysis and visualization. It has a client-server architecture and a data-flow-driven execution model [9]. Data Explorer has been implemented for Unix workstations from Sun, Silicon Graphics, IBM, Hewlett-Packard, DEC and Data General, and personal computers using OS/2.

Client-Server Architecture

The client process in this package is the graphics user interface. It utilizes X Window and the Motif window manager and is implemented in C++. The server process

operates as a computational "engine" and is implemented in C. It may reside on the same or different systems than the client. The server is controlled via a data-flow executive, which determines what tasks need to be executed based upon user requests and schedules their execution. The server accepts a well-defined protocol (a scripting language), which is generated by the user interface. The executive can be operated independently of the user interface via that scripting/programming language.

Uniform Data Model

One of the design criteria for Data Explorer was adaptability to new applications and data, and the utilization of multiple types of data simultaneously. Another was efficiency for access among the functions that a user might employ. Both of these requirements have been addressed by building the module set on a foundation of an integrated, discipline-independent data model [5]. The implementation of this data model describes and provides uniform access services for any data brought into, generated by, or exported from the software for a number of interesting classes of scientific, engineering and graphics data, which can be described by shape (size and number of dimensions), rank (e.g., scalar, vector, tensor), type (float, integer, byte, etc. or real, complex, quaternion), where the data are located in space (positions), how the locations are (topologically) related to each other (connections), mesh dependency of data (i.e., node or cell center), nodes or cells that may be invalid, user-defined metadata or aggregation (e.g., hierarchies, series, polylines, composites, multi-zone grids). It also supports those entities required for graphics and imaging operations within the context of Data Explorer (e.g., viewing camera, normals for shading, etc.).

All operations on data within Data Explorer, independent of a role in generating pictures, work with shared data structures in memory via an uniform interface that is presented by the data model. This permits the same consistent access to data independent of its underlying grid, type or hierarchical structure(s). To minimize copying and reduce memory utilization, data communication among subsequent operations is accomplished by passing pointers. In addition, sharing of these structures among such operations is supported.

One result of this approach is easy integration of disparate, multiple data sets, a requirement becoming more common for many visualization problems (e.g., results of observation and simulation, remote sensing from space and ground truth, differing medical imaging modalities, structural analysis, fluid flow and design data). This integration can take place without unnecessary conversion or interpolation operations that would corrupt the data. An example of this idea is shown in figure 4, where a number of distinct atmospheric cloud parameters are shown simultaneously in a three-dimensional, earth-centered coordinate system.

Visual Programming and Polymorphism

An important consequence of the unified method of data handling is that operations in Data Explorer (modules) are polymorphic, interoperable and appear typeless to the user. This is in contrast to other available implementations, in which each class of supported data is handled more or less independently and is utilized with a separate set of modules.

Consider figures 5 and 6, which show a very simple example of visual programming with Data Explorer. They each contain a screen dump of the Visual Program Editor with a 4-node network. The available modules are shown in various categories on the left. A number of options associated with creating and manipulating visual programs as well as interaction with other aspects of both the Data Explorer executive and user interface are available through pull-down menus from the top of the Visual Program Editor. The *Import* module reads specified data from a file or pipe. The *Isosurface* module computes surfaces of constant value. In this case, the third input has been specified with the number 4, which means that four surfaces at four equally spaced values over the range of data will be computed. The *AutoColor* module computes a linear hue-based color map (blue to red) over the full range of the data. The *Image* module renders an image from the input it receives and provides tools to interact with the rendered image.

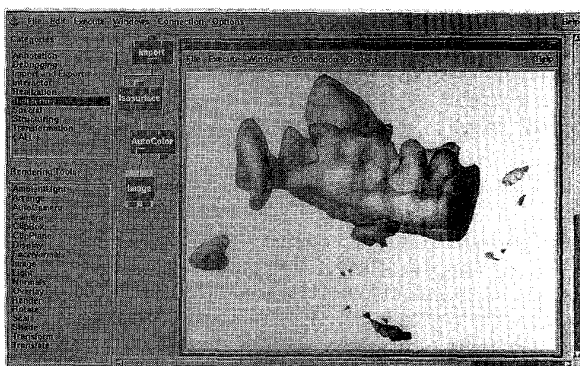


Figure 5. Simple Data Explorer Visual Program Applied to a Regular Three-Dimensional Data Set.

The image in figure 5 shows four isosurfaces computed from a three-dimensional stack of CAT scans of a human spine, comprising a regular, rectilinear volume of cubes. The image in figure 6 shows four contour lines computed from ultraviolet intensities observed from a spacecraft in a curvilinear, irregular grid of quadrilaterals. The polymorphic nature of the modules allows the same set of tools, and hence, the same visual program to be applied to disparate data sets without intervention by the user. For example, *Isosurface* computes surfaces from three-dimensional data, lines from two-dimensional data, and points from one-dimensional data, independently of the type of

mesh structure or space within which the data are embedded.

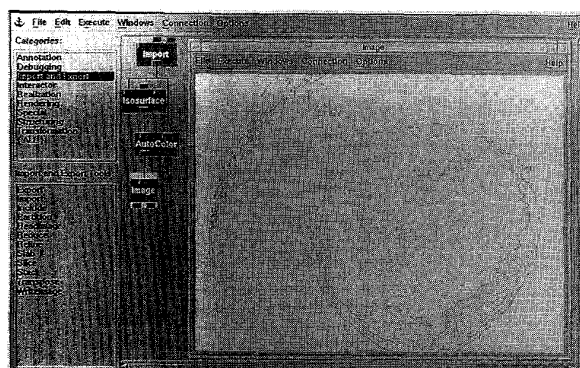


Figure 6. Simple Data Explorer Visual Program Applied to an Irregular Two-Dimensional Data Set.

Module Toolkit

The collection of polymorphic modules in Data Explorer provides various computational tools for the user. They support a number of realization techniques for generating renderable geometry from data. These include color and opacity mapping (e.g., for surface and volume rendering), isosurfaces, histograms, two-dimensional and three-dimensional plotting, surface deformation, etc. for scalar data. For vector data, arrow plots, streamlines, streaklines, etc. are provided. Realizations may be annotated with ribbons, tubes, axes, glyphs, text and display of data locations, meshes and boundaries. Data probing, object picking, arbitrary surface and volume sampling, and arbitrary cutting/mapping planes are supported. Data Explorer includes a number of non-graphical operations such as standard mathematical functions, univariate statistics and image processing. Field/vector operations such as divergence, gradient and curl, dot and cross products, etc. are provided. Non-gridded or scattered data may be interpolated to an arbitrary grid or triangulated. The length, area or volume of various geometries may also be computed. Tools for data manipulation such as removal of data points, subsetting by position, sorting, sub/supersampling, grid construction, mapping, interpolation, regridding, transposition, etc. are available. The data structures that support the data model may also be manipulated and queried at the module level.

Data-Flow Execution in Data Explorer

A number of problems associated with the data-flow execution of graphs produced by visual programming and their potential solutions have been discussed. The execution model of Data Explorer incorporates the two approaches (data cache and graph analysis) discussed earlier as well as several others.

Iteration: The Data Cache

As described earlier, efficient execution of visualization programs in an iterative context demands the retention of results of a module. Hence, if the inputs to the module have not changed on a subsequent execution of the graph, the result can be re-used without re-computation. Data Explorer extends this notion by incorporating a cache for all partial results. This cache retains results from not only the previous execution of the network, but from all prior executions. This is subject to memory limitations and a least-recently-used cache flushing strategy. Caching may also be explicitly set by a user for each output of each module to optimize memory utilization. For example, in figures 5 and 6, if one only wanted to keep the colored isosurface, then caching could be turned off explicitly for the upstream modules, *Isosurface* and *Import*. Using information stored in the cache, one can not only retrieve the results of parts of the network that have not changed from the previous execution, but can also return to previous states efficiently. If a module has executed with a given set of inputs at some time in the past, and one returns to those input settings, re-execution of the module may be avoided by finding the result in the cache.

This capability is particularly useful in conjunction with the Data Explorer *Sequencer* module. The *Sequencer* provides a very simple and flexible animation capability -- an automatic method of managing a frame counter in a graph, which is updated with each new execution based upon a VCR-like interactor. The *Sequencer* includes buttons for stop, pause, run forward and run backward. It also includes buttons to place it into single-step mode, to cause it to continually loop and to do so in "palindrome" mode. The settings window is used to specify the limits of the sequence of numbers generated.

The first time the *Sequencer* is "played", it will cause the network to be executed with new values for the *Sequencer* output. Each execution, which may be time consuming, will result in a new image being generated. These images, which after all, are simply the result of a rendering module, will be retained in the cache. When the *Sequencer* is "re-played", the inputs to the network are the same as they were for the first execution. Thus, the result of the execution (the images) will be immediately available from the cache. Hence, Data Explorer provides an automatic mechanism to create real-time animations even when the computation of each frame is slower than real-time. These features are illustrated in figures 7 and 8.

The value produced by the *Sequencer* can be used in a number of ways. Figure 7 shows how the *Sequencer* may be used to iterate through a time-dependent data set, causing the visualization to operate on each time step in turn, resulting in an animation showing how the data vary with time. The user may look at a daily sequence of one or more isosurfaces of atmospheric temperature over the earth's southern hemisphere as well as to interactively change the value(s) specified to the *Isosurface* module. The program also illustrates the support of a subroutine

hierarchy. Two of the tools, *Projections* and *WorldMapProjections*, are not atomic operations but macros, visual programs that include references to a number of modules and other macros. Alternatively, figure 8 shows how the *Sequencer* can be used to drive the iso-value input to the *Isosurface* module. Note how the *Statistics* and *Compute* modules are used to scale the integer *Sequencer* values to vary through the actual data range of the data. Polymorphism in the modules enables such a network to operate on any scalar data for any data type primitive.

The idea of storing results in cache can be extended to include inputs to interactors, which correspond to user interface widgets. Interactors used in this fashion are considered data-driven and thus, have state. Both figures 7 and 8 illustrate data-driven interactors. In figure 7, the data range is used to automatically determine default settings for the *ScalarList* interactor (e.g., minimum and maximum values). In this case, the output of the *ScalarList* interactor is used to set the values at which isosurfaces should be computed. In figure 8, information derived from statistics computed from the data is used to define the maximum number of frames that can be generated by the *Sequencer*.

Conditional Execution

Data Explorer incorporates two mechanisms to control execution flow through the network: the *Switch* and *Route* modules. *Switch* essentially implements a case construct, which has n+1 inputs: a selection value (from 1 to n) and n inputs of potential selections. *Route* is the inverse of *Switch*, with n outputs and two inputs: a selector value (which may be a list of integers) and an object to be passed to the selected n outputs.

As described earlier, implementing conditionals in data-flow systems efficiently requires that unnecessary paths through the network be skipped. In the case of *Switch*, only the required input (if one is selected) is evaluated. *Route* "kills off" the sub-networks depending on unselected outputs until a *Collect* module joins a result depending on an unselected output of the *Route* module with valid results. Figure 9 shows how *Route* can be used to allow the user to select a subset of visualization techniques from a set implemented in the network.

Data Explorer also permits a network to control the execution of aspects of the user interface such as control panels (e.g., open/close), execution mode, image windows (e.g., display mode), etc. internally, that ordinarily would require user action. Coupled with tools for conditional execution, portions of the interface can be made available or hidden based upon user input or computation.

External Asynchronous Data Sources

Many applications of visualization tools call for a direct interface with external data sources, especially ones that generate data to be studied (e.g., a computational simulation). The execution model of Data Explorer provides the

framework for real-time visualization of data generated asynchronously by such a process. An external data source is linked into a Data Explorer network by incorporating a communications module, which receives data from the external source, often across a socket, and passes the resulting data object to the module's output. This module (and its descendents) will only run when the external data source has indicated that new data are available.

Data Explorer also provides a mechanism for direct manipulation of the executive (e.g., mode, passing data, error handling, etc.) and the user interface (e.g., window visibility and mode, tracking mouse events, etc.) from an external application. This allows control of Data Explorer from other software and peer-to-peer communications.

Parallelism

The aggregation of all Data Explorer tasks representing a collection of computational "modules" are mapped to a single process with intratask parallelism. Under user control (i.e., within the user interface client), the server may be distributed such that arbitrary portions of a Data Explorer program may be specified to execute within a slave server(s) process operating on another networked system(s). Each of the server process(es) may contain any number of tasks as with the (original) master process. In addition, user-defined modules may be utilized via separate executables from the server process(es) or data may be accepted via a pipe from another process.

In principle, the use of parallelism is an effective way to improve performance. To achieve maximum benefit the system must provide near-linear speed-up as one adds processors. If the software only supports intermodule parallelism, which can be consistent with a distributed execution module, it may be very difficult to achieve efficient parallel execution even on a modest coarse-grain machine for more than a handful of processors. Intramodule parallelism is better suited to exploit such coarse-grain parallelism. Visualization is a complex operation for which the benefits of parallel execution may vary from problem to problem. It is therefore important that a visualization system provides both intermodule and intramodule parallel execution modes, as does Data Explorer. Intermodule parallelism is best suited for problems where two or more computationally intensive operations can execute independently on multiple processors. Linear speed-up may not occur in this case due to the speed of communication between nodes for passing data. It is however, the simplest mode of parallelism to implement, especially on clusters of workstations and distributed memory multiprocessor systems. Intramodule parallelism is best suited for the exploitation of shared-memory multiprocessor systems applied to problems which have a sequence of one or more computationally intensive operations. By enabling modules to execute in an intramodule manner, computation can be accelerated on multiprocessor machines. This method also obviates the need to pass large volumes of data across a network between multiple pro-

cessors. On symmetric multiprocessor systems, intramodule parallelism is supported through a simple fork-join shared-memory paradigm.

The Data Explorer executive process uses data domain decomposition and task scheduling. The data domain is partitioned by use of facilities in the data model whereby a single field can be split into a group of smaller spatially local self-contained regions (i.e., composite field). The boundaries of the sub-fields are "grown" to avoid boundary effects in subsequent realization operations. Each parallelized module generates sub-tasks to operate on each partition of the data. This approach also avoids the explicit use of locks, thereby reducing the possibility of a deadlock.

Preserving Explicit State

Some visualization applications require the retention of state from one execution to the next, which cannot be supported within the context of pure data-flow. Consider, for example, the creation of a plot of data values at a point while sequencing through a time series. In effect, the state of the plot is retrieved from the prior execution, appended with the new time-step results, the updated plot is produced and the results are preserved by re-saving the state for the next execution. This capability is provided via two modules: *Set*, which places an object into the cache, and *Get*, which retrieves objects from the cache. While *Get* and *Set* (cache) provides a simple mechanism for storing such state, they do give rise to the same difficulties outlined above for external data sources - the execution of *Get* depends on more than its inputs, it also depends on whether the saved object has been changed. Fortunately, the same solution can be utilized. Whenever *Set* executes, it flags its paired *Get* to be run on the next execution.

An application of *Get* and *Set* is illustrated in figure 10, which shows a visual program that access a year's worth of observations of global ozone one day at a time. The *Sequencer* is used to specify the days to be examined. One *Get* and *Set* pair is used, which is highlighted. For each time step, a contour is determined at a specified level, which is colored (blue to red) and labelled by the day of the year. The *Get* and *Set* pair holds the accumulated time series of annotated contours, which is appended for each day in the series. The results up to the current day are displayed as a geographic map, which shows the evolution of the contour.

This approach can also be used to support true looping inside a program, which is illustrated in figure 11. Atmospheric temperature data are read and sliced by latitude. A loop is then initiated via the *ForEachMember* module, which computes the mean value for each slice and accumulates it into a series. As with the previous example, *Get* and *Set* are used to store the series for each iteration, but inside the loop. When the loop is finished, which is signaled to two *Route* modules, two images are produced, one showing a pseudo-color image of the tem-

perature data with a map overlay, and the other a plot of the mean temperature in each latitude zone.

Extending Data-flow vs. Alternatives

Song and Golin [14] and Pang and Alper [10] have discussed the idea of using a fine-grain decomposition of computation specified through data-flow, instead of the more typical coarse-grain implementations. This shows promise as a more efficient way to use memory and computational resources for operations that may be done serially on data subsets (e.g., isosurface, rendering). Unfortunately, this approach does not appear to be practical for specification of applications requiring dozens or hundreds of individual tasks under conditional execution, where saving state may be desirable, parallelized implementation of specific operations, or operations that are not easily decomposed in a fine-grain manner (e.g., streaklines - flow integration across multiple time steps of an unsteady vector field). Conventional data-flow with the class of extensions discussed earlier appears to be more effective at addressing such problems. In contrast, this approach may be very useful for supporting simpler visualization and analysis tasks on small machines (e.g., PCs).

Hibbard et al [7] in VIS-AD offers the virtues of an uniform and extensible data model within a programming environment that provides for sufficient control of operations to build realistic applications. However, it operates at a lower-level than Data Explorer and other data-flow tool kits with only basic graphics, data structure and computational primitives. Although VIS-AD does provide the facilities available in the extended data-flow architecture of Data Explorer with greater flexibility, it is at a cost of a larger learning curve and greater effort to build complex visualization and analysis operations. But it does provide an easier mechanism than a traditional programming language to develop new algorithms because of its inherent data model and interactive graphics primitives.

Conclusions

Traditional implementations of a data-flow execution model are quite limited when applied to problems of realistic complexity. Fortunately, a number of extensions to such a model are practical way of resolving these difficulties while still preserving the virtues of "pure data-flow". Extensions such as graph evaluation, conditional execution and caching have been embodied in the IBM Visualization Data Explorer software package. Efforts are continuing to enhance the implementation of the execution model in Data Explorer in response to user requirements for data analysis and visualization.

Acknowledgements

The authors thank David Watson of the IBM UK Scientific Centre, and Kevin McAuliffe, David Wood and Richard Sefceka of the IBM Thomas J. Watson Research

Center for their thoughtful reviews of and suggestions for this paper.

References

- [1] Abram, G. and T. Whitted. "Building Block Shaders". **Computer Graphics**, 24, n. 4, pp. 283-288, August 1990.
- [2] Arvind and Brobst, S. "The Evolution of Dataflow Architectures from Static Dataflow to P RISC". **International Journal of High Speed Computing**, 5, n. 2, pp. 125-153, June 1993.
- [3] Burnett, M. M., M. J. Baker, C. Bohus, P. Carlson, S. Yang and P. van Zee. "Scaling Up Visual Programming Languages". **IEEE Computer**, 28, n. 3, pp. 45-54, March 1995.
- [4] Dyer, D. S. "A Dataflow Toolkit for Visualization". **IEEE Computer Graphics and Applications**, 10, n. 4, pp. 60-69, July 1990.
- [5] Haber, R., B. Lucas and N. Collins. "A Data Model for Scientific Visualization with Provisions for Regular and Irregular Grids". **Proceedings IEEE Visualization '91 Conference**, pp. 298-305, October 1991.
- [6] Haeberli, P. "ConMan: A Visual Programming Language for Interactive Graphics". **Computer Graphics**, 22, n. 4, pp. 103-111, August 1988.
- [7] Hibbard, W., C. R. Dyer and B. Paul. "Display of Scientific Data Structures for Algorithm Visualization". **Proceedings IEEE Visualization '92**, pp. 243-249, October 1992.
- [8] Kass, M. "CONDOR: Constraint-Based Dataflow". **Computer Graphics**, 26, n. 2, pp. 321-330, July 1992.
- [9] Lucas, B., G. D. Abram, N. S. Collins, D. A. Epstein, D. L. Gresh, and K. P. McAuliffe. "An Architecture for a Scientific Visualization System". **Proceedings IEEE Visualization '92**, pp. 107-113, October 1992.
- [10] Pang, A. and N. Alper. "Max & Match: A Construction Kit for Visualization". **Proceedings IEEE Visualization '94**, pp. 302-309, October 1994.
- [11] Rasure, J. and C. Wallace. "An Integrated Data Flow Visual Language and Software Development Environment". **Journal of Visual Languages and Computing**, 2, pp. 217-246, 1991.
- [12] Ribarsky W., R. Brown, T. Myerson, S. Smith and L. Treinish. "Object-Oriented, Dataflow Visualization Systems - A Paradigm Shift?". **Proceedings IEEE Visualization '92**, pp. 384-387, October 1992.
- [13] Silicon Graphics Computer Systems. **IRIS Explorer User's Guide**, Document 007-1369030, 1993.
- [14] Song, D. and E. Golin. "Fine-Grain Visualization in Dataflow Environments". **Proceedings IEEE Visualization '93**, pp. 126-133, October 1993.
- [15] Upson, C., T. Faulhaber, D. Kamins, D. Laidlaw, D. Schlegel, J. Vroom, R. Gurwitz and A. van Dam. The Application Visualization System: A Computational Environment for Scientific Visualization. **IEEE Computer Graphics and Applications**, 9, n. 4, pp. 30-42, July 1989.

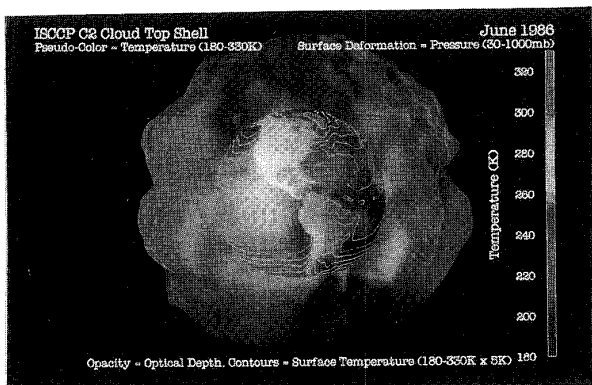


Figure 4. Several Atmospheric Parameters Shown Simultaneously.

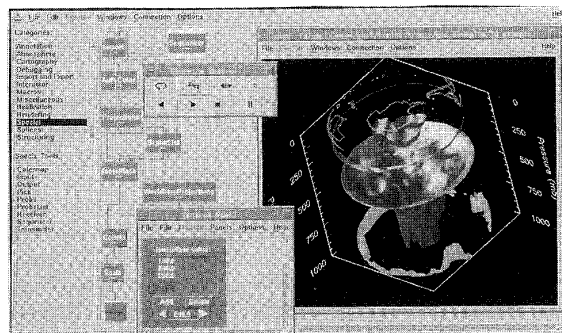


Figure 7. Adjustment of an Input Parameter and Sequencing through a Time Series.

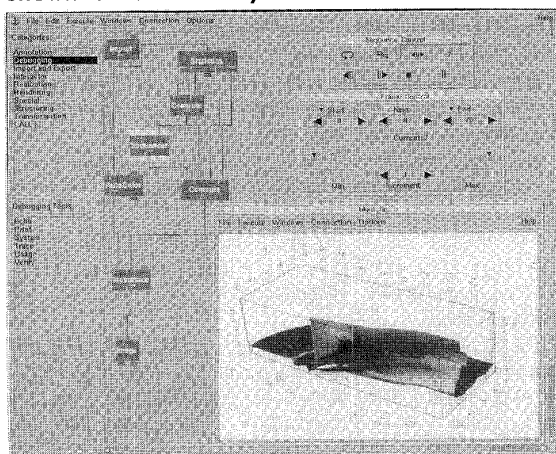


Figure 8. Use of the Data Explorer Sequencer for Iteration.

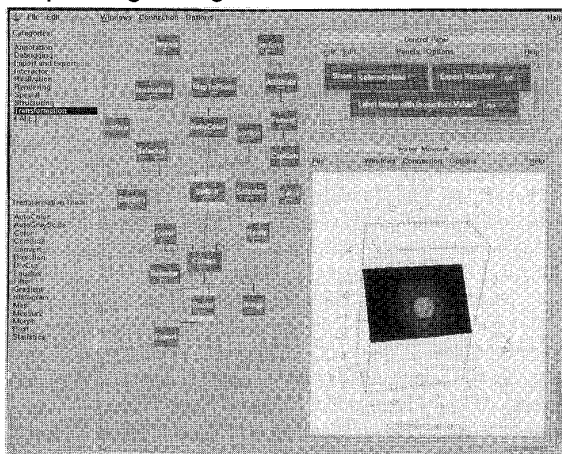


Figure 9. Flow Control in a Data Explorer Program Using the Switch and Route Modules.

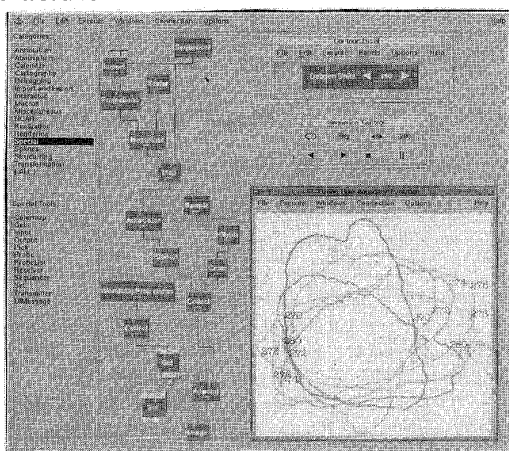


Figure 10. Preserving State in a Data Explorer Program Using the Get and Set Modules.

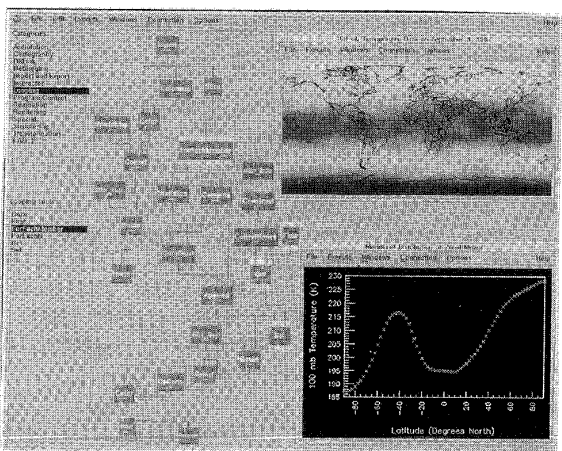


Figure 11. Looping in a Data Explorer Program.